

Implicit Actions and Non-blocking Failure Recovery with MPI

Aurelien Bouteiller
Innovative Computing Laboratory
The University of Tennessee
Knoxville, Tennessee, USA
Orcid: 0000-0001-5108-509X
Email: bouteill@icl.utk.edu

George Bosilca
Innovative Computing Laboratory
The University of Tennessee
Knoxville, Tennessee, USA
Orcid: 0000-0003-2411-8495
Email: bosilca@icl.utk.edu

Abstract—Scientific applications have long embraced the MPI as the environment of choice to execute on large distributed systems. The User-Level Failure Mitigation (ULFM) specification extends the MPI standard to address resilience and enable MPI applications to restore their communication capability after a failure. This work builds upon the wide body of experience gained in the field to eliminate a gap between current practice and the ideal, more asynchronous, recovery model in which the fault tolerance activities of multiple components can be carried out simultaneously and overlap. This work proposes to: (1) provide the required consistency in fault reporting to applications (i.e., enable an application to assess the success of a computational phase without incurring an unacceptable performance hit); (2) bring forward the building blocks that permit the effective scoping of fault recovery in an application, so that independent components in an application can recover without interfering with each other, and separate groups of processes in the application can recover independently or in unison; and (3) overlap recovery activities necessary to restore the consistency of the system (e.g., eviction of faulty processes from the communication group) with application recovery activities (e.g., dataset restoration from checkpoints).

Index Terms—message passing, fault tolerance, high performance computing, distributed systems

I. INTRODUCTION

Numerical simulations, data mining, and—most recently—machine learning have taken on critical roles in all fields of scientific exploration. High-performance computing (HPC) systems are specifically designed to provide the cutting-edge computing capabilities required to obtain timely answers to the crucial scientific challenges of our time, and—as such—are routinely breaking new barriers in harnessing a massive number of computing resources to deliver staggering levels of performance. Applications have widely embraced the Message Passing Interface (MPI) [1] to cope with the challenges posed by the deployment of scientific computation on distributed systems. That MPI is a standardized and stable specification, and that all HPC vendors provide an implementation tailored to deliver performance on their systems, ensures that users are confident that their application will easily port and achieve consistently high performance on current and future hardware.

Unfortunately, MPI lacks a comprehensive management system for failures, and resilience (or lack thereof) could

still pose a significant challenge for next-generation HPC systems [2], [3]. Since the Petascale era, reliability is already measured in days (e.g., production runs on Oak Ridge National Laboratory’s (ORNL’s) Titan system experience close to three failures per day [4]). The mean time between failures (MTBF) is dependent on the number of components in the system, and, despite expected major advances in the reliability of individual hardware components, and the prevalence of “fat” GPU nodes in currently deployed exascale systems (putting us in the optimistic part of pre-exascale predictions [5]), the MTBF for post-exascale systems could still decline significantly, or at the minimum, act as an external constraint that restricts machine design and component choices (and prices) to those that do can sustain an acceptable MTBF.

The HPC community has already started reacting to this threat and is actively investigating resilient applications and mitigation techniques that permit the efficient execution of scientific computing on machines that may experience multiple fault events per allocation. To accompany this community effort, we have designed the User-Level Failure Mitigation (ULFM) extension to the MPI specification [6] and have provided a reference implementation [7]. The ULFM specification provides the basic infrastructure to 1) detect and report failures of MPI processes; 2) interrupt the flow of the application; and 3) repair the state of MPI communication objects to restore the full communication capability (e.g., the ability to carry out collective communication). This early effort has clearly responded to a need in the community (see Section V and [8]), with multiple international research teams sustaining a two-pronged exploratory effort to investigate algorithmic fault tolerance techniques in fields as varied as physics, chemistry, and engineering on one hand, and a flurry of fault tolerance frameworks designed to ease and streamline the expression of resilience techniques on the other.

This paper explores how to again broaden the realm of possible methodologies for resilience techniques by exploring asynchronous and scoped techniques for error reporting and consistency of state recovery in the message passing layer. The overarching goal is to enable the concurrent recovery of multiple components of the software stack. Two major statements motivate this effort. First, the cost of recovering

the message passing’s full communication capability can be significant, and—in some instances—in the same order of magnitude as the cost of recovering the application state. Given the appropriate level of asynchrony, these complementary activities could overlap, thus significantly driving down the overall cost of recovery. Second, applications are typically composed of multiple software components in charge of different software functional modules. These modules can recover part of the application state independently, in a completely scoped insulation from the rest of the application, but may also require knowledge about a broader scope in order to trigger the recovery path at the appropriate moment and in the appropriate order to enable all modules to recover successfully. To expound further, this work explores how to: (1) provide the required consistency in fault reporting to applications, that is, enable an application to assess the potential success of a computational phase without incurring an unacceptable performance hit; (2) bring forward the building blocks that permit the effective scoping of fault recovery in an application so that independent components in an application can recover without interfering with each other, and separate groups of processes in the application can recover independently or in unison, as needed; (3) overlap recovery activities that are necessary to restore the consistency of the system (e.g., eviction of faulty processes from the communication group) with application recovery activities (e.g., dataset restoration from checkpoints); and (4) deliver these novel capabilities in a mature software environment (Open MPI) to ensure an easy path forward for application communities interested in fault tolerance research.

The rest of this paper is organized as follows: in Section II presents the background for MPI failure recovery and general ULFM concepts; Section III-A presents a technique to control the scope, that is, the set of processes and MPI operations, that get notified when a failure occurs; Section III-B discusses a new way of reporting errors in collective operations that simplifies a coordinated response to the failure; Section III-C presents a non-blocking MPI state rebuilding routing that enables more asynchrony and ease ordering constraints during recovery; Section IV presents the experimental results evaluating these concepts; Section V presents related works for MPI recovery systems, as well as discuss the applicability of the proposed ideas to a large set of fault tolerant applications and frameworks that currently use ULFM; and finally Section VI concludes.

II. BACKGROUND

Research in fault management techniques has a long and exceptionally rich history [9]. In some instances, there is a possibility that a hardware failure could result in drastic, life or death consequences. In mission-critical applications, when safety trumps performance, solutions such as duplication, triplication, were a long time favorite, as they permit detecting and mitigating faults at a high but predictable production and/or operation cost. In contrast, HPC considerations often call for a less drastic skew in the tradeoff between safety

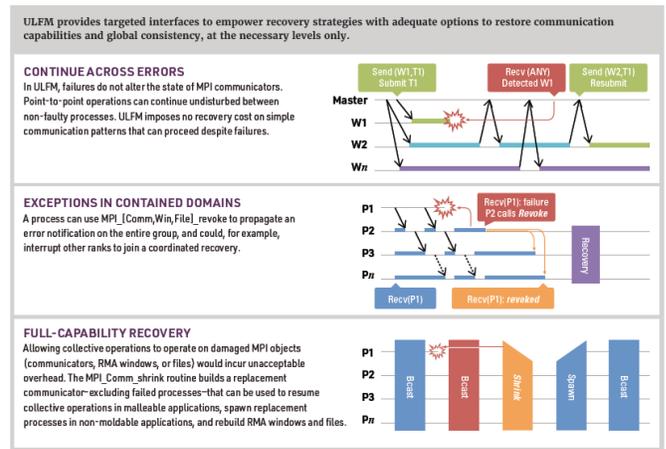


Fig. 1. Major use-cases for the ULFM MPI fault tolerance extension.

and performance. Although scientific computation requires a high level of confidence in the accuracy of the result produced, hardware failures (i.e., when a failure results in some processes of the application terminating unexpectedly) often manifest by preventing the completion of the application rather than corrupting the end results. Even for soft errors (i.e., when a failure results in a corruption of the computation accuracy or correctness), the problem is often reframed as an efficiency tradeoff, with the additional constraint that detecting the error may require periodic and extensive data invariant verification [10], [11] or an increase in per-operation cost to provide online error detection [12]. But once the failure has been asserted, recovery techniques are often very similar to those deployed in crash-failure scenarios, with restarting from a known good state (checkpoint) often being an adequate strategy [13], [14]. Thus, what users are generally seeking in HPC is the sweet spot where the cost of fault mitigation is lower than the potential downtime, delay, and duplicate computation cost incurred by the crash of the application [15]. Historically, global checkpointing [16]–[19] has been the mitigation technique of choice. However, performance models predict that we be may on the verge of an era where—despite its relative simplicity—more advanced techniques (checkpoint based or otherwise) will yield better efficiency [15], [20], [21]. This fact has steered a significant effort from the HPC community in exploring tailored techniques for tolerating failures in applications on one hand, and in exploring abstractions, runtime systems, and programming models that can effectively support the execution of fault tolerant applications on the other hand.

The ULFM specification [6] has recently taken a central role in this community (see Section V). It permits the continued execution of MPI programs after a failure by providing three critical capabilities: (1) the detection of failures and reporting of errors; (2) the ability to reliably disseminate a signal to interrupt the communication flow of the application; and (3) a consensus operation that permits stabilizing the state of the application and resuming normal operation after

ERROR SCOPING

Adding per-communicator (window/file) control knobs for the application to control the scope of error reporting: set Info key `mpbx_error_scope` on a communicator to control which errors interrupt MPI calls.

- **"local"**: current ULFM behavior: report an error only when communicating with a failed peer (e.g., rcv from failed process, collective communication) **default, current ULFM**
- **"group"**: report errors (i.e., REVOKE) for a failure at any process with a rank in the comm/win/file (e.g., in rcv from an alive process in comm)
- **"global"**: report errors (i.e., REVOKE) for a failure anywhere in "universe"

ERROR UNIFORMITY

All processes partake in a collective operation, should they return an error in unison? Use sets info key `mpbx_error_uniform` on a communicator to control if error reports need to be uniform.

- **"local"**: errors reported as needed to **inform of invalid outputs** (buffers/comms) at the reporting rank (i.e., other ranks may report success); **default, current ULFM**
- **"create"**: if communicator/win/file creation operations (e.g., `comm_split`, `file_open`, `win_create`, `comm_spawn`) reports at a rank, it has reported the same `ERR_PROC_FAILED/REVOKED` **at all ranks**
- **"col"**: same as above, for all collective calls (including creates)

ASYNCHRONOUS ERROR RECOVERY

Error recovery is difficult to overlap, because MPI currently misses asynchronous dynamic processes constructs.

- Adding `MPI_COMM_SHRINK` to enable asynchronous failed processes exclusion
- Adding `MPI_COMM_SPAWN` (and `ICONNECT/IAcCEPT`) to enable asynchronous spare respawn (as well as many other non-ft application use cases)

SCOPING EXAMPLE:
Only rank4 reports as it is communicating with failed rank 5.



UNIFORMITY EXAMPLE:
An error is reported only at some leaf node in a broadcast topology with a failure.

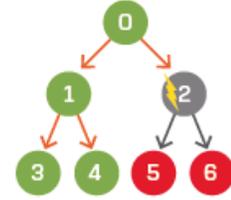


Fig. 2. New uniformity and scope modes in relation to existing ULFM definitions.

MPI internal structures and capabilities have been restored. In more details, following the diagrams in Figure 1, we can follow ULFM operating workflow by inspecting use-cases. Some applications can run-through faults, simply continuing to execute a point-to-point communication pattern that avoids communicating with known failed processes; typical for the class are manager-worker frameworks that simply resubmit the work delegated to a dead worker to another compute resource. For such applications, it is important that the effect of failure is the most localized, and that operation between non-failed processes continues normally. Some applications are malleable, that is, they require the full capabilities of MPI after recovery (e.g., collective communication) but can redistribute the work to execute on a smaller set of processes; such applications can use the `MPICH_COMM_SHRINK` operation to recreate working communicators that exclude dead processes. Finally, some applications have an inflexible mapping that requires an exact number of processes; the ULFM specification can be used in combination with MPI-2 dynamic operations, such as `MPI_COMM_SPAWN` to recreate an isomorphic communicator. Applications that are not run-through will often require a collective recovery procedure where all processes join to repair the MPI state, and the application dataset. New and original algorithms were developed to ensure the scalability of the resilience features in ULFM. In [22], we proposed a scalable and reliable broadcast algorithm to efficiently support the *revoke* operation. In [23], we proposed an original consensus algorithm tailored for HPC systems. In [24], we designed a scalable failure detector, and in [25], we studied replacement processes. With these developments, interest swelled among MPI implementors [26] and scientific communities as they

experimented with these capabilities to design fault tolerant frameworks and programming language extensions that support failure recovery, as well as algorithmic resilience methods over a range of applications. Feedback from early adopters has identified potential areas of improvement that we have set to explore in this paper.

III. A NOVEL APPROACH TO MPI RECOVERY

In the following subsections, we will describe the concepts we introduce to improve the reactivity, expressivity, and potential for overlap of recovery activities in MPI programs.

A. Error Scoping

Failure detection and subsequent error reporting is the starting point of the entire recovery procedure. In ULFM, the scope of error reporting is limited to processes that perform a direct communication operation with a failed process (Figure 2). The rationale is twofold. First, some application use cases operate under the assumption that as long as a can complete successfully (i.e., the group of processes participating in the operation does not contain a failed process) it must proceed without interruption. This approach is particularly tailored for job-stealing types of applications, where the failure of some workers is irrelevant to the operation of the remaining workers as well as to controller processes that do not manage these workers. Second, general failure detectors [27]–[30] can generate a significant number of background control messages, either when they require the establishment of pervasive heartbeat monitoring, or when employing gossip techniques. To preserve performance, the original ULFM specification was designed to enable an implementation where only in-band

error detection is provided (i.e., errors collected directly from the transport layer of MPI when the connection to an endpoint is broken at the network driver level).

However, prior results hint that: (1) the cost of out-of-band failure detection can be minimal and scalable when implemented correctly, as demonstrated by the performance of the failure-detection algorithm we designed for ULFM [31], and (2) when considering complex applications comprised on multiple software modules, certain modules may require more thorough information about errors that happen outside their direct neighbors in the communication pattern of that module; for example, it may be required for this component to be informed of any failure happening at any process on which this module spans, regardless of whether or not the module is communicating with this process at the moment.

The first aspect is to define the interface with which the MPI user can control the scoping behavior of the implementation to meet the desired criterion. Again, we added a control for the error-scoping behavior on a *per-communicator* basis, with the addition of MPI info keys that permit setting the specific desired behavior. That way application modules that require a different error scoping behavior can operate independently in different modes by posting the communication on different communicators. The modes we envision will permit: (1) reporting only errors when the posted communication operation cannot be completed, that is one of the peers participating in the communication has failed, and the resultant non-compliant behavior of the operation needs to be reported (this represents the current ULFM mode); (2) reporting an error when any process in the communicator is detected as failed, even if the failed process is not participating in reporting the MPI communication; and (3) reporting an error when any process failure in the MPI universe is detected. Given that the group and global reporting mode require a more stringent failure detector that can obtain out-of-band failure information, we identify the need to thoroughly investigate and compare the cost of these operating modes with the in-band only (i.e., from the network driver) detection cost.

One motivation for the group and global scope of error reporting is to ease the writing of fault tolerant applications. In many applications with a collective recovery pattern, ULFM users have to deploy a two-stage recovery process that first explicitly propagates the occurrence of a fault event (using the `MPPIX_COMM_REVOKE` operation) and only then proceeds to the collective recovery pattern. This explicit management is very flexible, but we expect the scoped recovery to significantly improve on the complexity of the error management code by streamlining multiple error paths (either an error is directly reported, or it has been triggered from an explicit propagation) into unified management.

Another critical motivation for the global scoping mode is enabling the cooperation of independent application modules or libraries that operate in the application. Although modules generally operate independently and require their communication to be segregated (i.e., employ separate communicators), many recovery techniques are global in nature, and a fault must

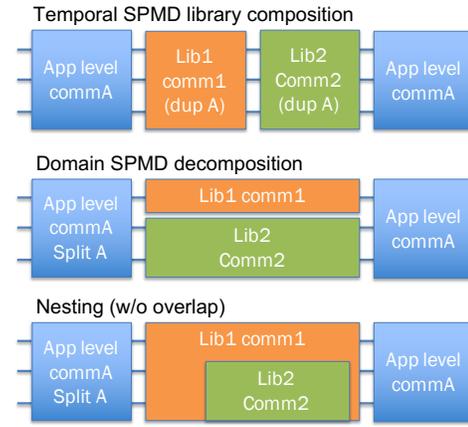


Fig. 3. Applications with multiple libraries may have processes using communicator handles that are not known by one of the libraries, calling for a method to observe faults at processes that are not part of the local communication pattern.

trigger a recovery action in all modules of the application—including in processes that may not be currently involved in communications for the module. Consider for example the case of a rank-domain decomposition (Fig. 3) where the application subdivides some ranks to work with library 1, while the rest of the ranks work with library 2. If the application requires a coordinated recovery action, both library modules must be interrupted; however, the library 1 does not have access to the handles for the communicators of library 2, and cannot directly revoke them. A solution is for both libraries to observe the global scope, which will interrupt implicitly both libraries when a fault occurs in any of the domains. Listing 1 demonstrates how to setup the library communicator in such a mode so that the library can detect faults at processes that are not currently engaged with the library-specific communicator (i.e., they may communicate in a separate communicator when the fault manifests).

Listing 1. Example of a library observing faults at processes that are not currently communicating on the library-specific communicator.

```
int lib1_init(MPI_Comm world) {
    /* ... */
    MPI_Comm_rank(world, &rank);
    MPI_Comm_size(world, &size);
    /* create a subcommunicator with only 1/4
     * the processes */
    int color = (rank < size/4)? 1: MPI_UNDEFINED;
    MPI_Comm_split(world, color, rank, &subcomm);
    if(MPI_UNDEFINED != color) {
        MPI_Info_create(&info);
        MPI_Info_set(info, "mpix_error_range", "universe");
        MPI_Comm_set_info(subcomm, info);
        /* From now on, a failure at a process that is **not**
         * calling lib1 will still cause communications on
         * subcomm to report errors */
        /* ... */
    }
}
```

B. Error Uniformity

Collective communications in MPI are a powerful tool for expressing patterns where a group of processes participate in the same communication operation. By abstracting the intent of the communication pattern, the operation can then be

optimized by the MPI implementation using communication topologies that limit the number of active connections between peers and improve the operation’s latency, and pipelining techniques that improve the achievable bandwidth. Collective operations also help structure the code of the application: operations like the `MPI_BARRIER` are specifically indented for separating algorithmic epochs. A simplistic assumption postulates that such structuring features would translate intact into a regime where failures may disrupt the execution of the application. To understand why this usually entails an unacceptable overhead, consider the case of a typical implementation of the `MPI_BCAST` operation over a tree topology. In the normal, fault-free operation, as processes relay the message from the root toward the leaf processes, they can complete the broadcast operation successfully as soon as they have forwarded the contribution. Thus, if a descendant process were to fail before forwarding the broadcast message, its parents would not return an error, yet any process in the subtree rooted at the failed process would have to do so, since the message never reached them. Note that this problem is not specific to one-to-all patterns, but can happen symmetrically in all-to-one (and all-to-all) patterns. Thus, in order to return an error at all ranks *uniformly*, the processes would have to *agree* on the outcome of the call, which is an extra fault tolerant synchronization step that is not required by the semantic of the underlying communication. Thus, to protect fault-free performance, in ULFM, the collective communication errors are defined as non-uniform (i.e., processes may return a different error status for the same operation), and the `MPI_COMM_AGREE` operation provides an explicit means for users to synchronize in a fault tolerant fashion when the need arises.

We identify the potential for simultaneously improving the performance and simplifying the expression of common usage patterns. First, some MPI collective operations often require a strong validation when deployed in practice. For example, operations that create new communicators often need to be validated for global success, or global failure, immediately. Similarly, in many iterative algorithms, the reduction step that computes the termination criterion is an excellent verification and restart point for the application, leading to many users’ employing an *allreduce* operation immediately followed by an *agreement*. Thus, in this work, we propose to provide the user with a level of control on the uniformity of error reporting. We propose setting a *per-communicator* property (practically, by setting specific MPI Info keys on the communicator) that lets the user decide whether collective operations on the communicator should operate at maximum speed (non-uniform) or with implicit safety (uniform error reporting at all ranks). This control enables a distinction between pure communication operations (which are often critical for performance) and setup/management operations (like operations that create new communicators, or operations that change the logical epoch in the algorithm).

Our implementation of the concept adds an agreement to all collective operations (an operation that can be performed in

approximately $2\times$ the latency of a small message, non fault-tolerant *allreduce* [23]).

C. Asynchronous Recovery

Many applications need to restore the full communication capability of MPI before they can resume their computational activity after a failure. The `MPI_COMM_SHRINK` operation provides the operational construct that permits recreating a fully functional communication context in which not only select point-to-point communication but also collective communication can be carried. The core of the operation relies on agreeing on the set of failed processes that need to be excluded from the input communicator and then producing a resultant communicator with a well-specified membership of processes. The current definition of the shrink operation is blocking and synchronous, which limits the opportunity for overlapping its cost.

We introduce a new non-blocking variant of the shrink operation, `MPIX_COMM_ISHRINK(comm, newcomm, req)`. To support this operation, we designed a non-blocking variant of the consensus-like elimination of failed processes as well as the selection of the internal context identifier (a unique number that is used to match MPI messages to the correct communicator on which the operation was posted).

The expected advantages are multiple. Some applications may have to recover multiple communicators after a failure. A common approach is to shrink the largest communicator (e.g., `MPI_COMM_WORLD`), and then recreate the damaged communicators with non-fault tolerant constructs (e.g., `MPI_COMM_DUP`, `MPI_COMM_SPLIT`, etc.), and then validate the whole set of new communicators with `MPI_COMM_AGREE` on the large communicator. This is an effective approach when the derived communicators have the same, or similar number of processes in their respective groups. This can however be sub-optimal in applications that create small neighbor communicators with orders of magnitude fewer processes in their respective groups than in the overarching communicator. In such a situation, directly shrinking multiple time smaller communicators can be advantageous. With the availability of the non-blocking shrink, these multiple shrink operation have an opportunity to overlap one another.

The non-blocking shrink also gives an opportunity for overlap between the cost of rebuilding communicators with the cost of restoring the application dataset (e.g., reloading checkpoints, computing checksum on data, etc.).

D. Implementation

These three ideas are implemented in the ULFM reference implementation, which is currently integrated into the main OPEN MPI development branch and slated for distribution with OPEN MPI version 5.

A major new requirement with the introduction of the error reporting modes is that the implementation may not only use in-band error reporting. In-band error reporting (bottom pathway in Fig. 4 is when the network driver itself reports fault events, that get propagated through the software stack

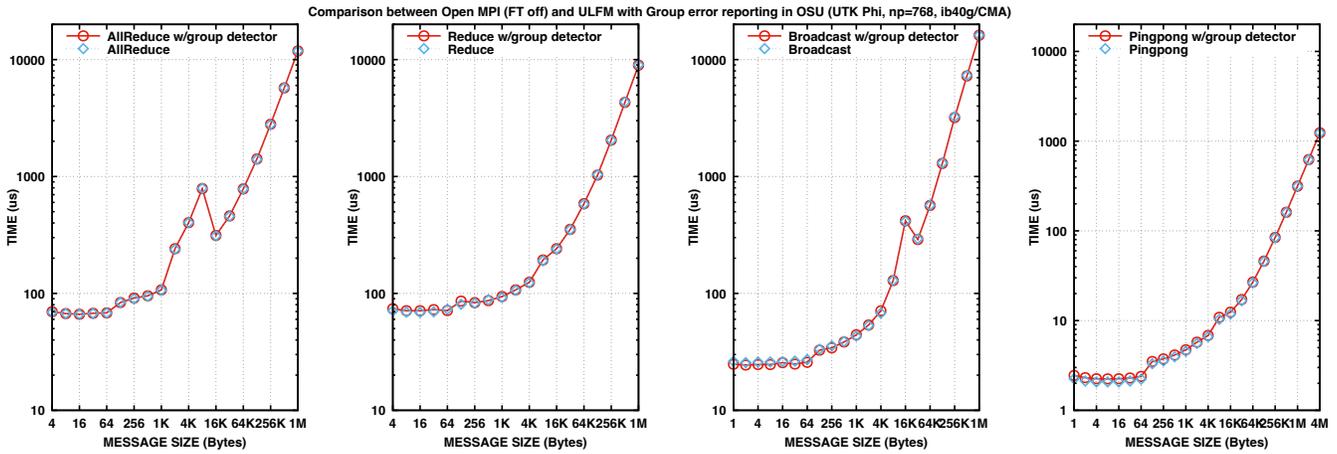


Fig. 5. Performance comparison between in-band only (Local scope) and out-of-band (Group scope) failure detection (768 MPI ranks)

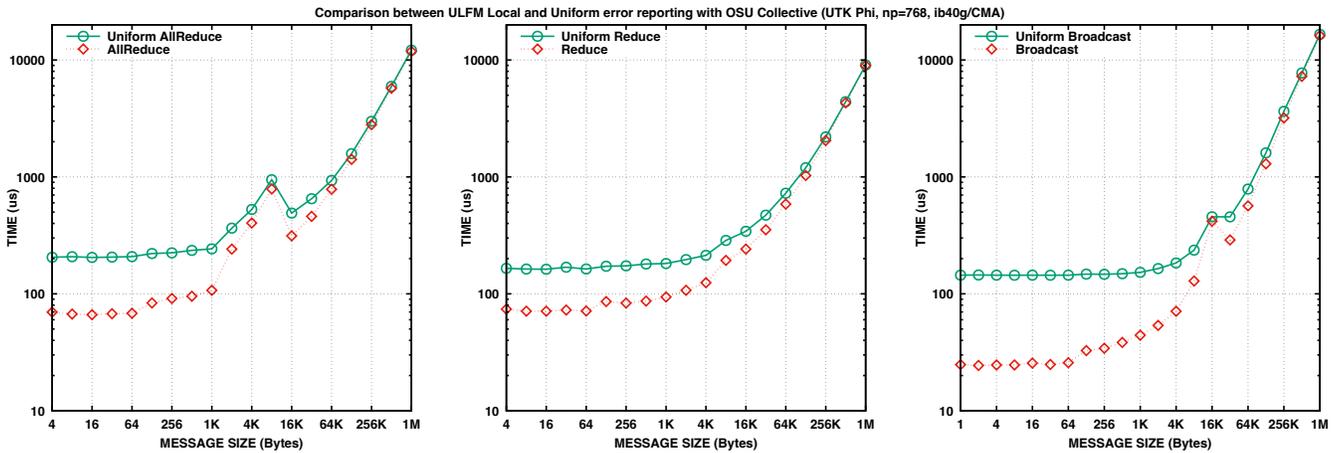


Fig. 6. Performance comparison between Local and Uniform collective communication (768 MPI ranks)

is). The benchmarks are selected to cover one-to-many, many-to-one, and many-to-many communication patterns. One can observe that in all cases, the addition of the verification step to enforce uniform error reporting has a significant impact on small message latency. In addition, in one-to-many and many-to-one operations, there is a significant departure in the inter-rank distribution of completion times: when the error reporting is local, some processes (presumably close to the root of the broadcast topology) complete the broadcast operation much more quickly than other processes (presumably leaves); in the uniform mode, the operation becomes synchronizing in a strong sense and all ranks experience a similar wait time for the broadcast operation to complete. In terms of bandwidth, the difference between the local and uniform modes decreases with message size, because the fixed cost of the verification step can be amortized with the increased cost of the communication itself.

E. Concurrent Shrinks Overlap

Fig. 7 presents the performance observed when rebuilding multiple communicators after a single rank process fault.

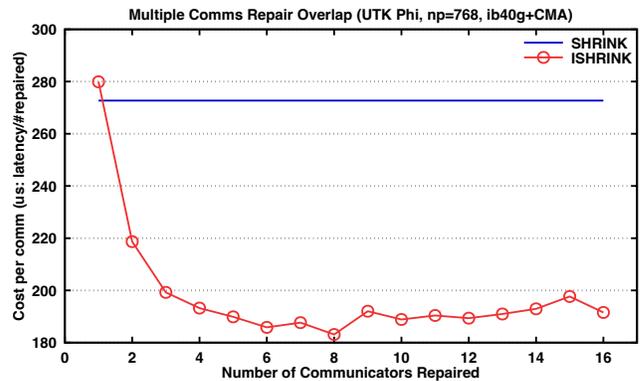


Fig. 7. Overlap between multiple concurrent ISHRINK operations (768 MPI ranks)

When using the non-blocking shrink operation, multiple communicators can be repaired concurrently. We present the per communicator cost, that is, the time it takes to repair the set of communicators, divided by the number of communicators.

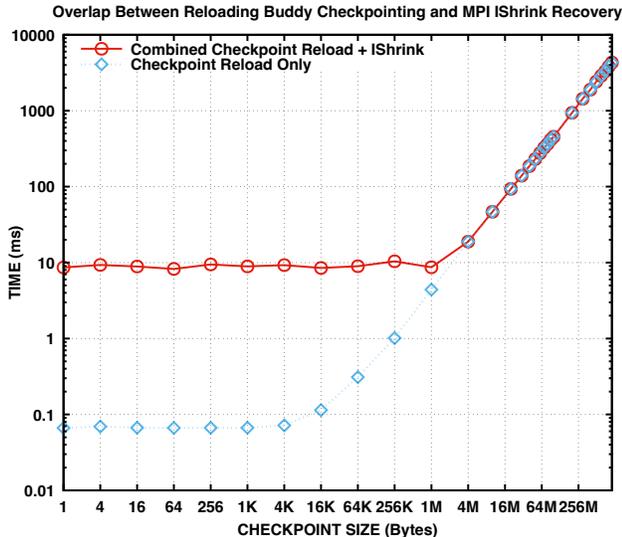


Fig. 8. Overlap between ISHRINK and Buddy Checkpointing Reload (768 MPI ranks)

The first observation is that using the blocking or non-blocking variants of the shrink operation takes approximately the same time to repair a single communicator; we observe only a small overhead from the split between initiation and completion in the non-blocking variant. When we increase the number of communicators that are repaired simultaneously (a pattern that can only be achieved using the non-blocking shrink variant), we observe that the cost per communicator decreases, up to approximately 8 communicators repaired simultaneously. This indicates that applications can reduce the time to recovery by overlapping the repair of multiple communicators with concurrent non-blocking shrink calls.

F. Shrink and Checkpoint Reload Overlap

Fig. 8 presents the time to recovery after a single rank failure when the application overlaps the time to reload a *buddy checkpoint* with the cost of repairing the communicators (as is needed when the application is intent on using collective communication after the recovery completed). We vary the size of the checkpoint dataset from very small (1 Byte) to large (1GB). At the scale of our experiments, the cost of performing a single shrink operation is very small (around $800\mu s$). The cost of repairing a communicator after a fault (thus including fault detection inside a shrink operation) is $10ms$. On this system, the cost of reloading the buddy checkpoint is equal to the cost of performing the shrink communicator repair operation between checkpoint sizes 1MB to 4MB. Of note, for the 4MB checkpoint, the time to reload a checkpoint is in the same order of magnitude as the cost of a shrink operation, yet the combined time is in-trend hinting that for this checkpoint size, the cost of the shrink is completely overlapped. The checkpoint sizes for which we achieve overlap without the sheer cost of the checkpoint amortizing the cost of a serial shrink are rather small on this system. However, larger scale

experiments carried in the past with ULFM show that, while the cost of buddy checkpointing is constant with the number of nodes (it varies with the per-node dataset size), the cost of performing a shrink operation increases with scale, which increases the importance of overlapping the shrink cost at scale, even for larger checkpoint loads.

V. RELATED WORKS

Other models of fault tolerance in MPI have also been researched. A number of projects deploy masking fault tolerance (i.e., failures are automatically handled by the environment, and MPI processes are replaced implicitly from replicates or checkpoints [16]–[19]). One detriment to masking fault tolerance is the typically high cost incurred on failure-free operation. FT-MPI [32] presented an earlier attempt at enabling fault management in MPI applications. FT-MPI automatically repairs the predefined `MPI_COMM_WORLD` communicator based on a replace, shrink, or blank policy. The shrink and replace modes are synchronous, and the approach is global in nature. MPI Reinit [33], [34] proposes a rollback model in which faulty processes are replaced automatically, all processes restart after the MPI initialization, and data restoration remains under the user’s control. By definition this approach is global and is fully synchronous. FMI [35] introduces a comprehensive model that handles fault tolerance, including checkpointing the application state, restarting failed processes and automatically reallocating additional nodes. Compared to ULFM—which proposes a flexible low-level API that supports a variety of fault tolerance models—these alternatives propose embracing a monolithic recovery model that supports a single mode of recovery, one that is always operating at a global scope.

For fault tolerance frameworks and languages, multiple frameworks that simplify checkpoint-restart, and user-controlled, in-place, global restart have been implemented on top of ULFM. For example, Fenix [4], [36] captures errors reported from ULFM, and rollback the application to the exit of the initialization function with repaired communicators (in shrink or replace mode). An explicit data management interface permits saving and restoring application data thereafter. In NR-MPI [37], failures are detected from the resource manager rather than from MPI operations, and NR-MPI takes care of replacing failed processes (using ULFM) and triggers data recovery procedures at the application level. In [38], the authors propose an alternative error reporting model based on operational timeouts and employ the ULFM implementation to prototype their effort. In the X10 language, an exception-based management of process failures is implemented on top of ULFM [39]. In the Co-Array Fortran language, the concept of “failed images” abstracts the detection and elimination of failed processes and is implemented using ULFM shrink operations [40]. In the Local Failure Local Recovery (LFLR) [41] framework, failure management is inserted in the application through C++ inheritance of recoverable data structures that require protection. The ULFM-based implementation captures MPI errors and substitutes spare processes

on which recoverable data is re-instantiated. In [42] the author provided a framework for placing checkpoints at semantic synchronization points. In [43], the authors survey different fault tolerance techniques in MPI and their applicability to the Fortran language. In [44], the authors study the cost of re-spawning processes and identify it as one of the largest overheads left standing on the recovery path. Some of these framework developers have expressed interest in the addition of asynchronous recovery of communicators as an important use-case for their workflow (e.g., Fenix, LFLR), and this emerging need has motivated the work presented in this paper.

On the application side, in [45], the authors study a fault tolerant algorithm for the resolution of 2-D stochastic Euler equations of gas dynamics with a monte-carlo method; in [46], the authors study numerical recovery strategies for Krylov solvers; in [47], the authors study sender-based message logging deployed at the application level; in [48], the authors study the requirements for deploying deep learning fault tolerance; and in [49], the authors present an application-based checkpoint compression strategy for finite-element multi-grid algorithms. Many of these Applications demonstrate usage patterns that recover multiple communicators, and recover small group communicators in isolation from one-another, a good fit for the group and local scoping, as well as non-blocking shrink overlap between communicator repairs.

Nontraditional HPC workloads (e.g., distributed database management systems [DDBMS] [50]) and big data analytics on the cloud [51]–[53] have also expressed interest in using MPI but had been deterred by its lack of resilience support. Using ULFM, they have obtained very impressive performance speedup and demonstrated that a fault-tolerant MPI can clearly serve some of their needs. In [54], the authors evaluated an MPI-based implementation of Hadoop and Map-Reduce with fault tolerance; in [55], the authors port the SAP database system over ULFM and demonstrate an impressive speedup when compared to system-level checkpoint and restart; the usage pattern would particularly benefit from the uniform collective mode of operation proposed in this paper. In [56], the authors consider a wish list of features required to execute cloud compute services over MPI and consider how to match their needs with ULFM features.

VI. CONCLUDING REMARKS

In the past, MPI fault recovery has considered only monolithic solutions, where either the entire application restarts, or the management of failures is totally transactional and contingent on the messaging pattern. In contrast, this work enable new patterns for fault tolerance—patterns that do not rely on a blocking, serialized, chain of parallel steps but instead enable merging and overlapping multiple steps of the recovery.

We have performed an evaluation of the proposed ideas in a practical setting using a mature software environment. The performance demonstrate that enabling a flexible scoping in error reporting and asynchronous, implicit recovery actions does have the potential to reduce the cost, while at the same

time maintaining a level of performance indistinguishable from the non fault-tolerant implementation.

Although our proposal and evaluation is in the context of MPI, the concepts and designs are expected to carry over to other programming models (e.g., partitioned global address space models [57]) beyond a narrow message-passing classification. These new capabilities form a new basis upon which application researchers and system designers can investigate how asynchronous recovery can be deployed in their own application domain.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1664142 SI2-SSI: EVOLVE: Enhancing the Open MPI Software for Next Generation Architectures and Applications.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

- [1] M. P. I. Forum, “MPI: A Message-Passing Interface Standard,” <http://www.mpi-forum.org/>, September 2012.
- [2] P. Beckman, R. Brightwell, B. R. de Supinski, M. Gokhale, S. Hofmeyr, S. Krishnamoorthy, M. Lang, B. Maccabe, J. Shalf, and M. Snir, “Exascale Operating Systems and Runtime Software Report,” US Department of Energy, Technical Report, December 2012. [Online]. Available: <http://science.energy.gov/~media/ascr/pdf/research/cs/Exascale%20Workshop/ExaOSR-Report-Final.pdf>
- [3] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. Chien, P. Coteus, N. Debardeleben, P. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. Hensbergen, *Addressing Failures in Exascale Computing*. U.S. DoE, 2013. [Online]. Available: <http://www.osti.gov/scitech/servlets/purl/1078029>
- [4] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, “Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’14, 2014.
- [5] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichniewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, “The International Exascale Software Project roadmap,” *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011. [Online]. Available: <http://dx.doi.org/10.1177/1094342010391989>
- [6] W. Bland, A. Bouteiller, T. Héault, G. Bosilca, and J. Dongarra, “Post-failure recovery of MPI communication capability: Design and rationale,” *IJHPCA*, vol. 27, no. 3, pp. 244–254, 2013.
- [7] W. Bland, A. Bouteiller, T. Héault, J. Hursey, G. Bosilca, and J. J. Dongarra, “An Evaluation of User-Level Failure Mitigation Support in MPI,” in *19th EuroMPI*, J. L. Träff, S. Benkner, and J. Dongarra, Eds. Vienna, Austria: Springer, Sep. 2012.
- [8] N. Losada, P. González, M. J. Martín, G. Bosilca, A. Bouteiller, and K. Teranishi, “Fault tolerance of mpi applications in exascale systems: The ulfm solution,” *Future Generation Computer Systems*, vol. 106, pp. 467–481, 2020.

- [9] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, p. 212, 2009.
- [10] M. Fasi, J. Langou, Y. Robert, and B. Uçar, "A backward/forward recovery approach for the preconditioned conjugate gradient method," *Journal of Computational Science*, vol. 17, pp. 522–534, 2016, recent Advances in Parallel Techniques for Scientific Computing. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S187750316300461>
- [11] A. Schöll, C. Braun, M. A. Kochte, and H. Wunderlich, "Low-overhead fault-tolerance for the preconditioned conjugate gradient solver," in *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, Oct 2015, pp. 60–65.
- [12] D. Li, Z. Chen, P. Wu, and J. S. Vetter, "Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 44:1–44:12. [Online]. Available: <http://doi.acm.org/10.1145/2503210.2503226>
- [13] G. Bosilca, A. Bouteiller, T. Herault, Y. Robert, and J. Dongarra, "Composing resilience techniques: ABFT, periodic and incremental checkpointing," *International Journal of Networking and Computing*, vol. 5, no. 1, pp. 2–25, 2015.
- [14] M. Fasi, Y. Robert, and B. Uçar, "Combining backward and forward recovery to cope with silent errors in iterative solvers," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 980–989.
- [15] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. J. Dongarra, A. Guermouche, T. Héroult, Y. Robert, F. Vivien, and D. Zaidouni, "Unified model for assessing checkpointing protocols at extreme-scale," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 17, pp. 2772–2791, 2014. [Online]. Available: <https://doi.org/10.1002/cpe.3173>
- [16] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine, "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI," in *IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.
- [17] A. Bouteiller, T. Héroult, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V project: a multiprotocol automatic fault tolerant MPI," vol. 20. SAGE Publications, Summer 2006, pp. 319–333.
- [18] K. M. Chandy and L. Lamport, "Distributed snapshots : Determining global states of distributed systems," in *Transactions on Computer Systems*, vol. 3(1). ACM, February 1985, pp. 63–75.
- [19] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," in *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.
- [20] T. Davies, C. Karlsson, H. Liu, C. Ding, , and Z. Chen, "High Performance Linpack Benchmark: A Fault Tolerant Implementation without Checkpointing," in *Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011)*. ACM.
- [21] A. Bouteiller, T. Héroult, G. Bosilca, P. Du, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy," *ACM Trans. Parallel Comput.*, vol. 1, no. 2, pp. 10:1–10:28, Feb. 2015. [Online]. Available: <http://doi.acm.org/10.1145/2686892>
- [22] A. Bouteiller, G. Bosilca, and J. J. Dongarra, "Plan B: interruption of ongoing MPI operations to support failure recovery," in *Proceedings of the 22nd European MPI Users' Group Meeting, EuroMPI 2015, Bordeaux, France, September 21-23, 2015*, 2015, pp. 11:1–11:9.
- [23] T. Héroult, A. Bouteiller, G. Bosilca, M. Gamell, K. Teranishi, M. Parashar, and J. Dongarra, "Practical scalable consensus for pseudo-synchronous distributed systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, 2015, pp. 31:1–31:12.
- [24] G. Bosilca, A. Bouteiller, A. Guermouche, T. Héroult, Y. Robert, P. Sens, and J. J. Dongarra, "Failure detection and propagation in HPC systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, J. West and C. M. Pancake, Eds. IEEE Computer Society, 2016, pp. 312–322. [Online]. Available: <https://doi.org/10.1109/SC.2016.26>
- [25] A. Hori, K. Yoshinaga, T. Héroult, A. Bouteiller, G. Bosilca, and Y. Ishikawa, "Sliding substitution of failed nodes," in *Proceedings of the 22nd European MPI Users' Group Meeting, EuroMPI 2015, Bordeaux, France, September 21-23, 2015*, 2015, pp. 14:1–14:10.
- [26] W. Bland, H. Lu, S. Seo, and P. Balaji, "Lessons learned implementing user-level failure mitigation in MPICH," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*. IEEE Computer Society, 2015, pp. 1123–1126. [Online]. Available: <https://doi.org/10.1109/CCGrid.2015.51>
- [27] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, Mar. 1996.
- [28] C. Leangsuksun, T. Rao, A. Tikotekar, S. Scott, R. Libby, J. Vetter, Y.-C. Fang, and H. Ong, "Ipmi-based efficient notification framework for large scale cluster computing," in *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 2, May, pp. 23–23.
- [29] R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, ser. Middleware '98. London, UK, UK: Springer-Verlag, 1998, pp. 55–70. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1659232.1659238>
- [30] A. Das, I. Gupta, and A. Motivala, "Swim: Scalable weakly-consistent infection-style process group membership protocol," in *International Conference on Dependable Systems and Networks*, Washington, DC, USA, 2002, pp. 303–312.
- [31] G. Bosilca, A. Bouteiller, A. Guermouche, T. Héroult, Y. Robert, P. Sens, and J. J. Dongarra, "A failure detector for HPC platforms," *IJHPCA*, vol. 32, no. 1, pp. 139–158, 2018. [Online]. Available: <https://doi.org/10.1177/1094342017711505>
- [32] Graham E. Fagg and Edgar Gabriel and George Bosilca and Thara Angskun and Zhizhong Chen and Jelena Pjesivac-Grbovic and Kevin London and Jack J. Dongarra, "Extending the MPI specification for process fault tolerance on high performance computing systems," in *Proceedings of the International Supercomputer Conference (ICS) 2004*. Primeur, 2004. [Online]. Available: <http://www.netlib.org/utk/people/JackDongarra/PAPERS/isc2004-FT-MPI.pdf>
- [33] S. Chakraborty, I. Laguna, M. Emani, K. Mohror, D. K. Panda, M. Schulz, and H. Subramoni, "Ereinit: Scalable and efficient fault-tolerance for bulk-synchronous mpi applications," *Concurrency and Computation: Practice and Experience*, vol. 0, no. 0, p. e4863, e4863 cpe.4863. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4863>
- [34] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, B. R. de Supinski, K. Mohror, and H. Pritchard, "Evaluating and extending user-level fault tolerance in mpi applications," *The International Journal of High Performance Computing Applications*, vol. 30, no. 3, pp. 305–319, 2016. [Online]. Available: <https://doi.org/10.1177/1094342015623623>
- [35] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. d. Supinski, N. Maruyama, and S. Matsuoka, "Fmi: Fault tolerant messaging interface for fast and transparent recovery," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1225–1234. [Online]. Available: <http://dx.doi.org/10.1109/IPDPS.2014.126>
- [36] M. Gamell, R. F. Van der Wijngaart, K. Teranishi, and M. Parashar, "Specification of the Fenix MPI Fault Tolerance library, version 1.0.1," Sandia National Laboratories, Livermore, CA, Tech. Rep. SAND2016-10522, October 2016.
- [37] G. Suo, Y. Lu, X. Liao, M. Xie, and H. Cao, "Nr-mpi: A non-stop and fault resilient mpi," in *2013 International Conference on Parallel and Distributed Systems*, Dec 2013, pp. 190–199.
- [38] A. Hassani, A. Skjellum, R. Brightwell, and P. V. Bangalore, "Comparing, contrasting, generalizing, and integrating two current designs for fault-tolerant MPI," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 63:63–63:68. [Online]. Available: <http://doi.acm.org/10.1145/2642769.2642776>
- [39] S. S. Hamouda, J. Milthorpe, P. E. Strazdins, and V. Saraswat, "A resilient framework for iterative linear algebra applications in X10," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, 2015, pp. 970–979.

- [40] J. Reid, "Additional CoArray features in Fortran," in *Proceedings of the 7th International Conference on PGAS Programming Models*. The University of Edinburgh, 2013, pp. 104–110.
- [41] K. Teranishi and M. A. Heroux, "Toward local failure local recovery resilience model using MPI-ULFM," in *21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, Kyoto, Japan - September 09 - 12, 2014*, 2014, p. 51.
- [42] N. Losada, I. Cores, M. J. Martín, and P. González, "Resilient MPI applications using an application-level checkpointing framework and ULFM," *The Journal of Supercomputing*, vol. 73, no. 1, pp. 100–113, 2017. [Online]. Available: <https://doi.org/10.1007/s11227-016-1629-7>
- [43] N. Weeks, G. Luecke, P. Maris, and J. Vary, "Challenges in developing mpi fault-tolerant fortran applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2018, pp. 524–531.
- [44] M. Szpindler, "Enabling adaptive, fault-tolerant MPI applications with dynamic resource allocation," in *Proceedings of the 3rd International Conference on Exascale Applications and Software*, ser. EASC '15. Edinburgh, Scotland, UK: University of Edinburgh, 2015, pp. 53–57. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2820083.2820094>
- [45] S. Pauli, P. Arbenz, and C. Schwab, "Intrinsic fault tolerance of multilevel monte carlo methods," *Journal of Parallel and Distributed Computing*, vol. 84, pp. 24–36, 2015. [Online]. Available: <http://www.scopus.com/inward/record.url?eid=2-s2.0-84938393594&partnerID=40&md5=ec6d7d72b0aafb8b91c50d239b937639>
- [46] E. Agullo, L. Giraud, A. Guermouche, J. Roman, and M. Zounon, "Numerical recovery strategies for parallel resilient Krylov linear solvers," *Numerical Linear Algebra with Applications*, vol. 23, no. 5, pp. 888–905, 2016, nla.2059. [Online]. Available: <http://dx.doi.org/10.1002/nla.2059>
- [47] J. Ahn, "N fault-tolerant sender-based message logging for group communication-based message passing systems," 2015, pp. 1296–1301. [Online]. Available: <http://www.scopus.com/inward/record.url?eid=2-s2.0-84925238874&partnerID=40&md5=8fdaad1dff202068f593e87bf10e75a1>
- [48] V. Amatya, A. Vishnu, C. Siegel, and J. Daily, "What does fault tolerant deep learning need from mpi?" in *Proceedings of the 24th European MPI Users' Group Meeting*, ser. EuroMPI '17. New York, NY, USA: ACM, 2017, pp. 13:1–13:11. [Online]. Available: <http://dx.doi.org/10.1145/3127024.3127037>
- [49] D. Göttsche, M. Altenbernd, and D. Ribbrock, "Fault-tolerant finite-element multigrid algorithms with hierarchically compressed asynchronous checkpointing," *Parallel Comput.*, vol. 49, no. C, pp. 117–135, Nov. 2015. [Online]. Available: <https://doi.org/10.1016/j.parco.2015.07.003>
- [50] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: It's time for a redesign," *Proc. VLDB Endow.*, vol. 9, no. 7, pp. 528–539, Mar. 2016.
- [51] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf," *Procedia Computer Science*, vol. 53, pp. 121 – 130, 2015.
- [52] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards efficient MapReduce using MPI," in *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 240–249. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03770-2_30
- [53] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "DataMPI: Extending MPI to Hadoop-like big data computing," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 829–838.
- [54] Y. Guo, W. Bland, P. Balaji, and X. Zhou, "Fault tolerant mapreduce-mpi for hpc clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 34:1–34:12. [Online]. Available: <http://doi.acm.org/10.1145/2807591.2807617>
- [55] J. Stengler, "Fault tolerant collective communication algorithms for distributed database systems," Ph.D. dissertation, Technische Universität Darmstadt, 2017.
- [56] J. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi, "Using MPI in high-performance computing services," 2013, pp. 43–48. [Online]. Available: <http://www.scopus.com/inward/record.url?eid=2-s2.0-84886302666&partnerID=40&md5=fd4e8f2c423525ab5e3b2bf630e67cb2>
OpenSHMEM for Hybrid Environments - Third Workshop, OpenSHMEM 2016, Baltimore, MD, USA, August 2-4, 2016, Revised Selected Papers, ser. Lecture Notes in Computer Science, M. G. Venkata, N. Imam, S. Pophale, and T. M. Mintz, Eds., vol. 10007. Springer, 2016, pp. 66–81. [Online]. Available: https://doi.org/10.1007/978-3-319-50995-2_5